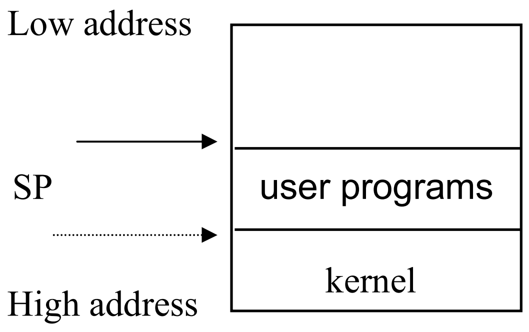
What is going on when a procedure is calling itself

```
entry main.enter;                 $RA       $PC
IMPORT …
block main uses proc {
const{
…
}
data{
…
}
code{
public enter:
    mov $S2, $A0;
    bsr proc1;
    …
}
}
block proc1 uses proc {
    code{
    public enter:
    addq $S0, $A0;
    bsr proc1;
    mov $V0,
    ret;
    }
    }
```

The stack

Sometimes functions need local memory in which values of local variables will be stored. Arrays have to be stored in memory, as they are often too big to fit in a register. Variables passed as reference parameters also need to be stored in memory, because they need to have an address where to refer. Local memory may also be used for saving the values of registers, the function uses for other purposes.
For assembly language programs, the memory is organized as a stack as shown in the diagram below.
The stack grows from high memory locations towards low memory locations.
Integer register 30, denoted as $SP (stack pointer), always points to the address of the top element in the stack.



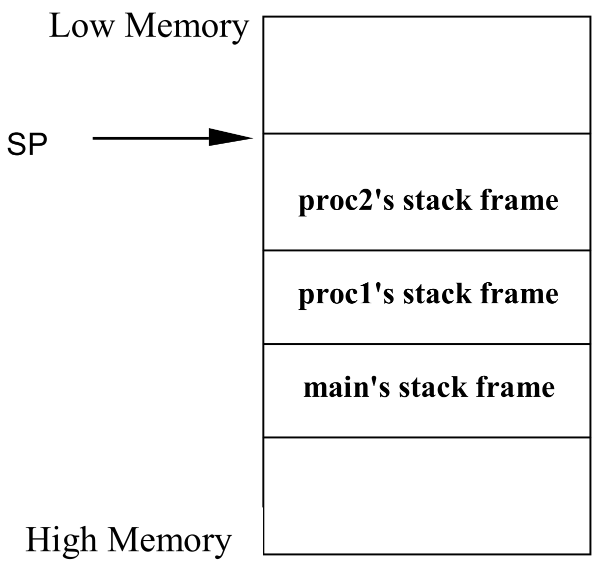
When a procedure is called, the procedure **may** first change the value of $SP to reserve some locations on top of the stack.
The locations are called **the stack frame** of the procedure.
The stack frame is used by the procedure to store the values of the registers whose values need to be preserved during a procedure call.
When a procedure terminates, the procedure should remove the stack frame from the top of the stack.
As the stack in grows towards low memory:

- Space on the stack is allocated by subtracting a constant number of bytes from the stack pointer register.
- Space is de-allocated by adding this constant number of bytes back onto the stack pointer register.
- Space for local variables can be accessed by using a non-negative displacement from the stack pointer register.

People usually think of the stack growing “up”. Therefore, when drawing diagrams, low addresses will usually be displayed at the top of the page, and high addresses at the bottom.

Procedure calls can be nested. Therefore, several stack frames might exist on top of the stack during the execution of a program. For example, the execution of the following program will make the stack as below

```
entry main.enter;
IMPORT …
block main uses proc {
const{
…
}
data{
…
}
code{
public enter:
    …
    bsr proc1;
    …
}
}
block proc1 uses proc {
    code{
    public enter:
    …
    bsr  proc2;
    …
    ret;
    }
    }
    …
block proc2 uses proc {
    code{
    public enter:
    …
    …
    ret;
    }
    …
```



Detailed usage of Registers

Here are the detailed conventions on how to use registers to guide you through function invocations.

The program counter register.
It points to the address of the next instruction to execute. It is always longword aligned.

32 integer registers.

0 v0 Used to hold the return value of an integer function. It may be altered by the invoked function, even if it doesn't return a result (for example, because the invoked function may invoke another function that returns a result).

1-8 22-25 t0-t7 t8-t11 These are temporary registers used for expression evaluation within a simple statement. They are not normally used to store data between statements.
These registers may be altered by the invoked function, so important data cannot be left in these registers while another function is invoked.

9-14 s0-s5 "Saved" registers, usually used to hold the values of local variables (in particular, local variables declared as "register variables" in C). If the invoked function wishes to use these registers, the invoked function **must** save the registers in its activation record on entry, and restore them on exit. As a consequence, the invoker can behave as if the invoked function never altered the registers.

15 fp Used to hold the frame pointer (address of the base of the activation record/stack frame/call frame) if needed. In most situations, the address of the activation record is the same as the top of stack, so the **sp** register can be used as the base address of the activation record, rather than the **fp** register, and the **fp** register is never set up. If a function needs to dynamically allocate local space in addition to its activation record (for example, for a dynamically sized local array), it can set the **fp** register to the base of the activation record, then allocate further space by further decrementing the sp register. The invoked function is responsible for saving and restoring this register.

16-21 a0-a5 These registers are used to pass the first six integer type actual parameters. The action of setting up these registers for the invoked function overwrites the values of the parameters for the invoker. A function that invokes another function must save the values of these registers on entry, either in its activation record, or in saved registers.

26 ra Used to hold the return address of a function. The program counter is saved in this register by the *bsr* instruction and restored from this register by the *ret* instruction. A function that invokes another function must save the value of this register in its activation record on entry, and restore it before exit.

29 gp Used to hold the global pointer. The global pointer points to a table containing the values of constants, such as the addresses of functions and global variables. This table is needed because the number of bits used to represent a constant in an instruction are too few to represent a 64 bit value. The ldiq pseudoinstruction is converted into a ldq instruction. The constant is stored in the global table, and accessed as an offset from the global pointer. The global pointer register is set up by the operating system when the program is loaded, and stays constant throughout the execution of the program.

30 sp This register is used to hold the stack pointer (the address of the "top" of stack). The invoked function allocates space for itself on the stack by subtracting the size of the activation record from the stack pointer. On return, the invoked function de-allocates the stack space by adding the size of the activation record to the stack pointer.

31 zero Always has the value 0.

Conventions for declaring functions on the Alpha

- If the function needs any local memory, allocate space for the activation record on the stack by subtracting the number of bytes needed from the stack pointer.
    - This is usually done by the instruction “lda $sp, -frameSize($sp);”.
    - “frameSize” is the size of the frame required in bytes.
        - It is a multiple of 8
- If the function invokes another function, save the return address register in the activation record of the function. (This is because invoking another function will overwrite the return address register.)
- If the function allocates space for local arrays, save the frame pointer register in the activation record of the function.
- If the function wants to make use of the “saved” registers for its own local variables, or saving the arguments, save these registers in the activation record of the function.
- If the function invokes another function, save the argument registers in “saved” registers. (This is better than saving them directly in the activation record, because heavy use is normally made of the arguments, and it is faster to access them via registers than from memory.)
- Temporary registers can be used, without need to save and restore them.
- Evaluate the body of the function. Use the saved registers for simple local variables that can fit into registers, and are not referred to by “reference”. Use memory on the stack for local arrays, etc.
- Store the return value in register v0.
- Restore any registers that were saved on entry to the function.
- If the function allocated any local memory for an activation record, deallocate this space by adding the size of the space to the stack pointer. This is usually done by the instruction “lda $sp, +frameSize($sp)”.
- A ret (return) instruction is used to return to just after the bsr instruction used to invoke the function.

For convenience, we define a block, proc, with a local section with symbolic names for the offsets for the saved values of the ra, fp, s0, s1, s2, s3, s4, and s5 registers. Because a register contains 8 bytes, the offsets saveRet, savFP, sav0, sav1, sav2, ... are 0, 8, 16, 24, 32, ...

```
block proc uses register {
//-----------------------------------------------------------
A local section is used to define the offsets for fields of records, activation
records of functions, etc. It does not allocate static space. It is primarily used
to generate the values of symbols representing the offsets of fields from the
base of the record, and the offsets of saved registers and local variables from
the base of the activation record.
/-----------------------------------------------------------
local {
protected savRet: quad;
protected savFP: quad;
protected sav0: quad;
protected sav1: quad;
protected sav2: quad;
protected sav3: quad;
protected sav4: quad;
protected sav5: quad;
protected sav6: quad;
}
}

block f uses proc {
code {
public enter:


//-----------------------------------------------------------
// Entry Code
//-----------------------------------------------------------
lda $sp, -frameSize($sp); // Allocate space on stack
stq $ra, savRet($sp); // Save ra on stack
stq $s0, sav0($sp); // Save s0 on stack
stq $s1, sav1($sp); // Save s1 on stack
... // ...
//-------------------------------------------------------
// Initialisation of variables
//-------------------------------------------------------
init:
mov $a0, $s0; // Save a0 in s0
mov $a1, $s1; // Save a1 in s1
// (Saving A registers only needed if your procedure invokes
// another function)
...
//-------------------------------------------------------
// Body of function
//-------------------------------------------------------
body:
...
mov ..., $v0; // Store result in v0
//-------------------------------------------------------
// Exit Code
//-------------------------------------------------------
end:
...
ldq $s1, sav1($sp); // Restore s1
ldq $s0, sav0($sp); // Restore s0
ldq $ra, savRet($sp); // Restore ra
lda $sp, +frameSize($sp); // Deallocate space on stack
ret;
}
}
```

/-----------------------Identifiers-------------------------------/

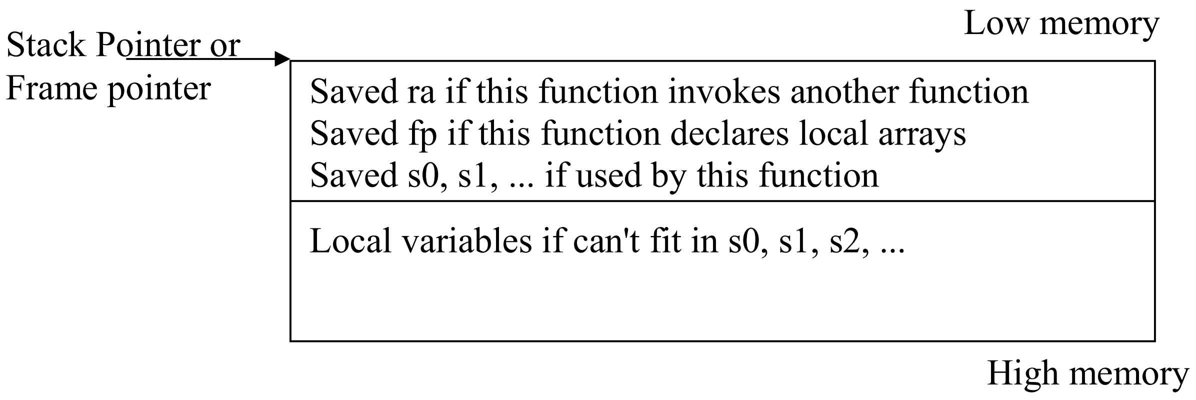
Identifiers can be declared as having public, protected, or private access. The default access is private.

An identifier declared within a section of a block can be referred to by its simple name anywhere within the block, including sub-blocks and compound statements.

An identifier declared within a block or named compound statement as public can be accessed outside the block or compound statement, by prefacing it by the name of the block or compound statement. Private and protected identifiers cannot be referred to in this manner.

An identifier declared within another block as public or protected, can be referred to in a block that uses it, by its simple name. Private identifiers cannot be referred to in this manner.

Layout of the activation record (call frame, or stack frame):


Low memory
Stack Pointer or
Frame pointer
Saved ra if this function invokes another function
Saved fp if this function declares local arrays
Saved s0, s1, ... if used by this function
Local variables if can't fit in s0, s1, s2, ...
High memory

Some portions of the activation record may be omitted.

- For simple functions only the portion used to save registers is likely to exist.
- Even this may be omitted for "leaf" functions that do not invoke other functions, and do not use the saved registers.

The activation record must be padded to a **multiple of 8 bytes**, so that all data is quadword aligned.

There are two types of procedures (programs).
One is leaf procedure, and the other is non-leaf procedure.

A leaf procedure does not call any other procedure.

- If the leaf procedure does not have any local variables, then there is not need to create the stack frame for the procedure.
- If the leaf procedure uses local variables then you need to create a stack frame.

A non-leaf procedure calls other procedures during its execution.

- A non-leaf procedure should always have a stack frame.

Since register `RA` is set to a new value when a `bsr` instruction is executed,
A non-leaf procedure should:

(a) Save the value of `RA` before it executes any `bsr` instructions.
(b) Restore the `RA` before it executes `ret` instruction.

Leaf procedure

A leaf procedure does not call any other procedure.
Not need to save the return address ($RA)

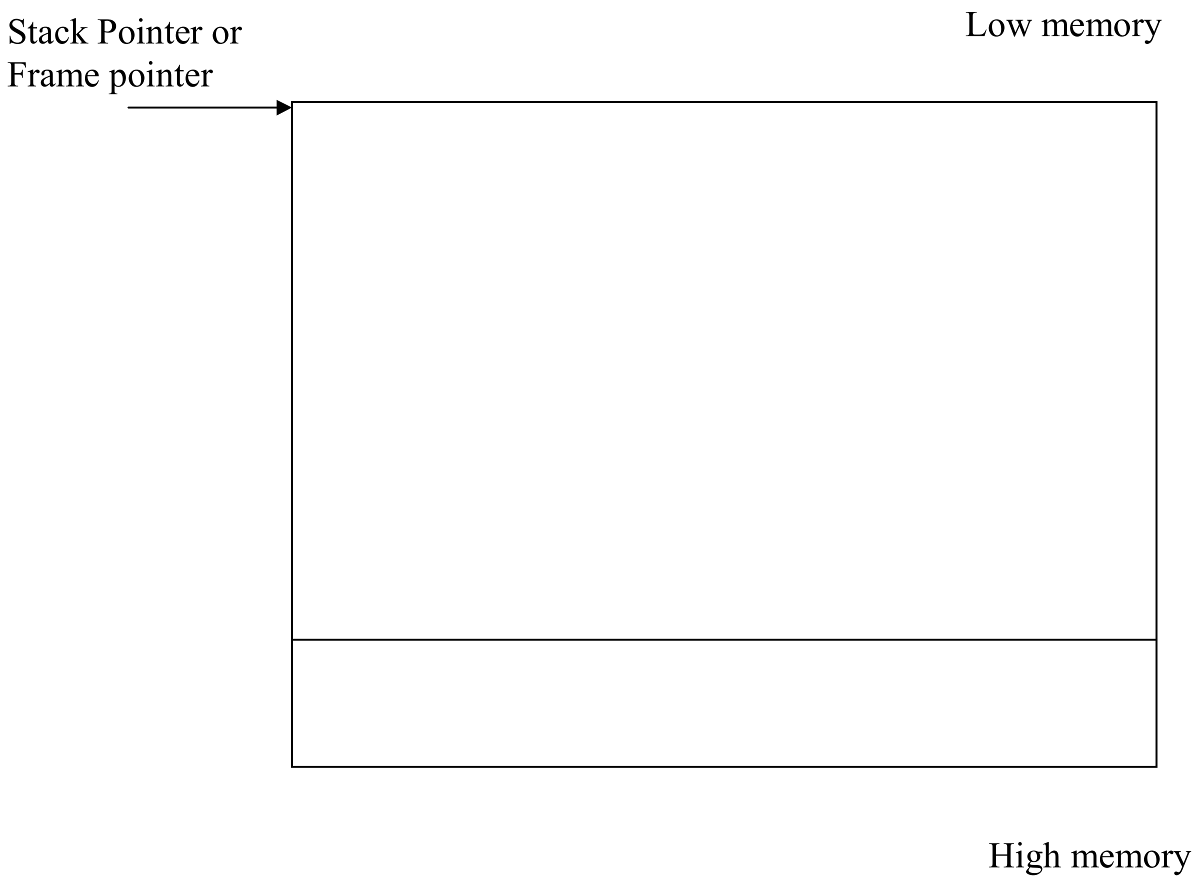
If the leaf procedure does not have any local variables, then there is not need to create the stack frame for the procedure.

- Use $A0 to $A5 to passed arguments.
- Use $V0 to return value.

```
Block main uses proc {
        data{
        ….
        }
        code{
        public enter:
        ….
        bsr procedure;
        ….
        }
}
        block procedure uses register {
        code{
                public enter:
                …
                ret;
                }
        }
```

```
//      void readLine( char *s, long max ) {
```

```
//              register long i = 0;
//              register long c;
//              while ( TRUE ) {
//                      c = getchar();
//                      if ( c == '\n' )
//                              break;
//                      if ( i < max )
//                              s[ i ] = c;
//                      i++;
//                      }
//              if ( i < max )
//                      s[ i ] = '\0';
//              else
//                      s[ max ] = '\0';
//              }
// 2 arguments are passed to this function (a pointer to an array of bytes, a maximum
// value). The function uses 3 local variables.
public block readLine uses proc {
     abs {
          s         =    s0;
          max       =    s1;
          i         =    s2;
          }
     code {
     public enter:
               lda       $sp, -sav3($sp);
               stq       $ra, savRet($sp);
               stq       $s0, sav0($sp);
               stq       $s1, sav1($sp);
               stq       $s2, sav2($sp);
     body:
          .
          .
          .
     return:
          ldq       $s2, sav2($sp);
          ldq       $s1, sav1($sp);
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav3($sp);
          ret;
               }
          }
```


Stack Pointer or
Frame pointer
Low memory
High memory